A Parallel Implementation of Tensor Multiplication

Bryan Rasmussen

Los Alamos National Laboratory

30 November 2006

Goals & requirements

Goal

To develop a parallel version of tensor multiplication with reductions.

Requirements

- At a minimum, multiply two rank-4 tensors with two reductions.
- Have potential for multiplying large tensors with applications in computational chemistry.
- Use memory efficiently.
- Scale well.

Basic definition

- **Tensor:** Extension of the idea of a linear operator to multi-linear algebra setting.
- Useful for writing equations with respect to arbitrary coordinate systems—many applications.

Just as we may write a linear operator as a matrix (in finite-dimensional space),

$$A = \begin{pmatrix} A\mathbf{e}_1 & A\mathbf{e}_2 & \cdots & A\mathbf{e}_m \end{pmatrix},$$

we may also write tensors as *multi-dimensional* boxes of numbers. Number of dimensions of box = *rank* of tensor, e.g.,

$$\text{rank-4 tensor:} \qquad a_{ijkn}.$$

Notation

Multiplication like an outer product.

$$c_{ijkmnp} = a_{ijk} b_{mnp}$$

Repeated indices $\Rightarrow$ summation

$$c_{ijmn} = a_{ij\color{red}{k}} b_{mn\color{red}{k}} \qquad \Leftrightarrow \qquad c_{ijmn} = \sum_k a_{ijk} b_{mnk}.$$

Also known as *tensor contraction*, or *reduction*.
In general, if we are multiplying $c_{**\ldots*} = a_{**\ldots*} b_{**\ldots*}$,

$$(\text{Rank } c) = (\text{Rank } a) + (\text{Rank } b) \text{ - } (2 \times \text{reductions}).$$

Notation examples

Consider 3-D column vectors **u**, **v**, and matrices A, B.

- Inner product: $s = u_i v_i = \sum_{i=1}^{3} u_i v_i = \mathbf{u}^T \mathbf{v}$
- Outer product: $w_{ij} = u_i v_j$

$$w_{ij} = u_i v_j = \begin{pmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \end{pmatrix} = \mathbf{u}\mathbf{v}^T$$

- Matrix-vector multiplication: $v_i = a_{ij} u_j$

$$v_i = a_{ij} u_j = \sum_{j=1}^{3} a_{ij} v_j = \begin{pmatrix} a_{11} v_1 + a_{12} v_2 + a_{13} v_3 \\ a_{21} v_1 + a_{22} v_2 + a_{23} v_3 \\ a_{31} v_1 + a_{32} v_2 + a_{33} v_3 \end{pmatrix} = A\mathbf{u}$$

Notation examples (continued)

- Matrix-matrix multiplication: $c_{ik} = a_{ij} b_{jk}$

$$c_{ik} = a_{ij} b_{jk} = \sum_{j=1}^{3} a_{ij} b_{jk} = AB =$$

$$\begin{pmatrix} \begin{pmatrix} a_{11} b_{11} + a_{12} b_{21} \\ + a_{13} b_{31} \end{pmatrix} & \begin{pmatrix} a_{11} b_{12} + a_{12} b_{22} \\ + a_{13} b_{32} \end{pmatrix} & \begin{pmatrix} a_{11} b_{13} + a_{12} b_{23} \\ + a_{13} b_{33} \end{pmatrix} \\ \begin{pmatrix} a_{21} b_{11} + a_{22} b_{21} \\ + a_{23} b_{31} \end{pmatrix} & \begin{pmatrix} a_{21} b_{12} + a_{22} b_{22} \\ + a_{23} b_{32} \end{pmatrix} & \begin{pmatrix} a_{21} b_{13} + a_{22} b_{23} \\ + a_{23} b_{33} \end{pmatrix} \\ \begin{pmatrix} a_{31} b_{11} + a_{32} b_{21} \\ + a_{33} b_{31} \end{pmatrix} & \begin{pmatrix} a_{31} b_{12} + a_{32} b_{22} \\ + a_{33} b_{32} \end{pmatrix} & \begin{pmatrix} a_{31} b_{13} + a_{32} b_{23} \\ + a_{33} b_{33} \end{pmatrix} \end{pmatrix}$$

Exercise: Construct AB^T, $\mathbf{u}^T A\mathbf{v}$, etc.

Application of interest

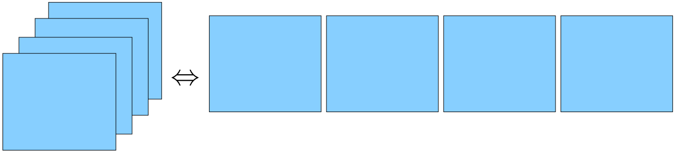
- Concentrate on $a_{ijkt} = b_{ij\,ef}\, c_{kt\,ef}$.
- Actually, code works for $w_{a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n} = u_{a_1 a_2 \ldots a_m c_1 c_2 \ldots c_p} v_{b_1 b_2 \ldots b_n c_1 c_2 \ldots c_p}$.
- Assume a k-index transformation:

$$v_{i_1 i_2 \ldots i_k} = \sum_{j_1 j_2 \ldots j_k = 1}^{M} z_{i_1 j_1} z_{i_2 j_2} \cdots z_{i_k j_k}$$

- Call z the *characteristic matrix*.
- Trade-off between storage and computation time.

Two ways to construct serial algorithm

- Our strategy: element-by-element multiplication.
  - Easier to read and analyze.
  - Easier to extend to arbitrary-rank, arbitrary-reduction.
- Another strategy: Unwrap the tensors.



  - Tensor operations become block-matrix multiplications.
  - Can use `BLAS` to compute.

Implementation in C++

`bryTensor` class description

Data

Tensor	Double-precision, not allocated until needed.
Char. matrix	Double-precision, not allocated until needed.
Statistics	Dimensions, ranks, tags, etc.
Mods	Cumulative products used for indexing.

Operations

Load/resize	Load tensor or characteristic matrix from file, `double *` variable, etc. Resize necessary for parallel version.
Formation	Form piece of tensor from characteristic matrix.
Product	Overwrites current tensor with product of two others. Arguments: pointers to tensors, # reductions.

Parallelization

Multiplying $w_{a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n} = u_{a_1 a_2 \ldots a_m c_1 c_2 \ldots c_p} v_{b_1 b_2 \ldots b_n c_1 c_2 \ldots c_p}$.

Assumption

- Each processor can hold one row of u, v and w.
- A *row*: `u(i,:,:,...,:)`.
- In index notation for rank-3 tensor: u_{2jk}.
- If u and v have 128 rows each, then each processor must be able hold $1/128^3 \approx 1/(2.1 \cdot 10^6)$ of problem.

Divide rows of u among processors, then divide rows of v among processors assigned to each row of u.

Notation: N_u, N_v are rows of u, v, respectively; P is number of processors.

$P < N_u$

- Each processor gets all of v.
- Each processor gets one or more rows of u.
  - If $N_u = 10$, and $P = 4$, then 2 processors would get 2 rows and 2 processors would get 3 rows.
- Start row of u assigned to processor n:

$$n \lfloor N_u/P \rfloor + \min\{n, (N_u \bmod P)\}$$

- Number of rows of u assigned to processor n:

$$\lfloor N_u/P \rfloor + \begin{cases} 1 & n < (N_u \bmod P) \\ 0 & \text{otherwise} \end{cases}$$

$P >= N_u$

- Each processor is assigned to one row of u.
- Each processor gets one or more rows of v.
  - If $N_u = 4$, and $P = 10$, then 6 processors would get $\approx 1/3$ of v, and 4 processors would get $\approx 1/2$ of v.
- Processor n is assigned to following row of u:

$$\text{row} = \begin{cases} n/(d+1) & n < m(d+1) \\ m + \dfrac{n - m(d+1)}{d} & n \geq m(d+1) \end{cases}$$

where $m = (P \bmod N_u)$, and $d = \lfloor P/N_u \rfloor$.

$P >= N_u$ (continued)

What piece of v does processor n get? Define

- Q: Number of processors on current row:

$$Q = \begin{cases} d+1 & n < m(d+1) \\ d & n \geq m(d+1) \end{cases}$$

- q: Rank of processor n in that list of processors

$$q = \begin{cases} n \bmod (d+1) & n < m(d+1) \\ [n - m(d+1)] \bmod d & n \geq m(d+1) \end{cases}$$

- Then the first row of v that processor n operates on is

$$q \lfloor N_v / Q \rfloor + \min \{q, (N_v \bmod Q)\}$$

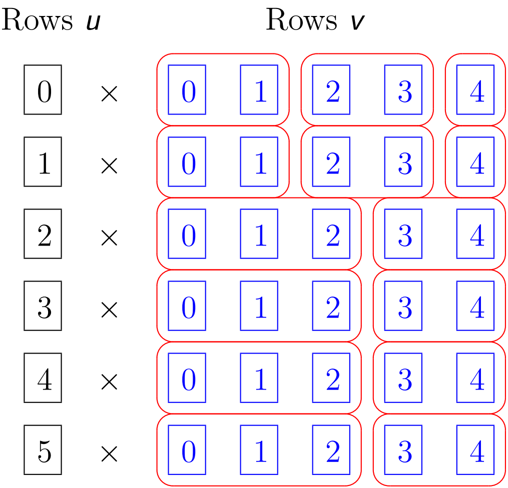
- The number of rows of v that processor n operates on is

$$\lfloor N_v / Q \rfloor + \begin{cases} 1 & q < (N_v \bmod Q) \\ 0 & \text{otherwise} \end{cases}$$

Distribution, $N_u = 6$, $N_v = 5$, $P = 4$

Rows u		Rows v				
$\boxed{0}$	$\times$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$
$\boxed{1}$	$\times$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$
$\boxed{2}$	$\times$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$
$\boxed{3}$	$\times$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$
$\boxed{4}$	$\times$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$
$\boxed{5}$	$\times$	$\boxed{0}$	$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$

Distribution, $N_u = 6$, $N_v = 5$, $P = 14$



Implementation issues

Consider two examples:

$P = 1000,\ N_u = 1000$

- Each processor is assigned to 1 row of u
- Each processor operates on all of v.

$P = 1999,\ N_u = 1000$

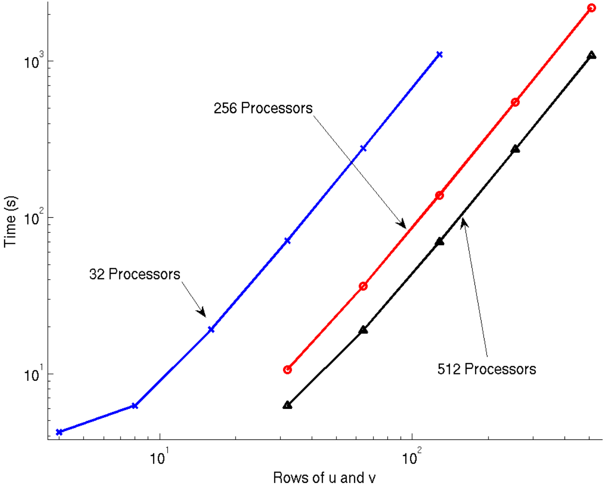
- Each processor is assigned to 1 row of u.
- Each of 1998 processors operates on $\approx 1/2$ of v.
- One processor operates on *all* of v.

We have basically doubled the number of processors, but computation time is the same! Moral of the story:

- If $P \geq N_u$, increase P by multiples of N_u.
- If $P < N_u$, increase P by integer divisions of N_u.

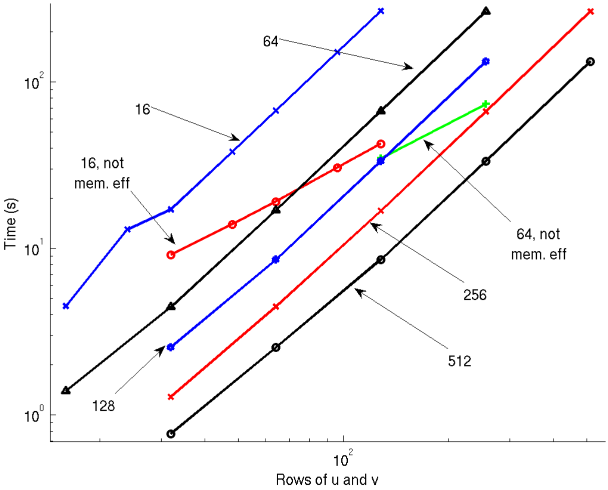
Timing, $u = y \times 32 \times 32 \times 32$, $v = y \times 32 \times 32 \times 32$

Exponents ≈ 1.99

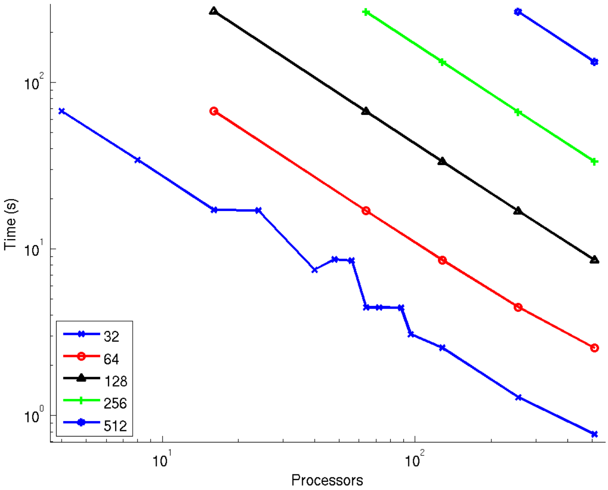


Timing, $u = y \times 16 \times 16 \times 16$, $v = y \times 16 \times 16 \times 16$

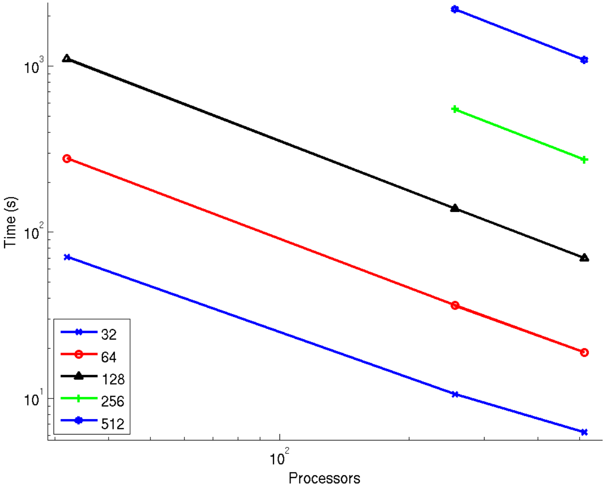
Exponents ≈ 1.99 and 1.11



Timing, $u = y \times 16 \times 16 \times 16$, $v = y \times 16 \times 16 \times 16$



Timing, $u = y \times 32 \times 32 \times 32$, $v = y \times 32 \times 32 \times 32$



Comments

Current algorithm

- Pretty good memory savings.
- Pretty good general algorithm.
- Lots of useful serial and `MPI` functions.
- Application will dominate storage/communication.
  - What do we do with this beast?
- Divergent behavior between memory-efficient mode and non-memory efficient mode when $P > N_u$.
- All in all, scales very well.

Comments (continued)

Future work

- Look at symmetry in k-index transformation:

$$v_{i_1 i_2 \ldots i_k} = \sum_{j_1 j_2 \ldots j_k = 1}^{M} z_{i_1 j_1} z_{i_2 j_2} \cdots z_{i_k j_k}$$

- Unwrap tensors and use the `BLAS`?
- Consider more optimal splitting strategy.
- Augment `MPI` calls with threads for better serial performance.
- Extend to different orders of indices.